QuickSort & Probabilistic Analysis

Suhas Arehalli

COMP221 - Spring 2024

1 QuickSort

- Much like MergeSort, QuickSort is a sorting algorithm that uses the divide and conquer design principle
  - Remember that divide and conquer algorithms solve a problem of size n by breaking them down into problems of size $< n$ (and as we've seen, hopefully a small fraction of n) and then finding a way to build a solution for the problem of size n from the smaller problems.
  - In MergeSort, we immediately turned a sorting problem of size n into two sorting problems of size $\frac{n}{2}$ (each sorting half the array), and then we constructed MERGE in order to build a solution of size n from those sorted halves.
  - For QuickSort, we'll do things in the reverse order: We'll modify the structure of the array of size n such that sorting two subarrays will sort the full array.
- Remember that if A is sorted, then for any $1 \leq k \leq N$,

$$A[1] \leq \cdots \leq A[k-1] \leq A[k] \leq A[k+1] \leq \ldots A[N]$$

  From this framing, we can define a weaker concept than sorting: We can say that an element $A[k]$ *partitions* A if for any $1 \leq i < k$, $A[k] \geq A[i]$, and for $k < j \leq N$, $A[k] \leq A[j]$.

  Crucially, if $A[k]$ partitions A, we know $A[k]$ will remain at index k in sorted order. Why? Because elements $< A[k]$ are to the left and should remain to the left and elements $> A[k]$ are on the right and should remain on the right! If elements are equal to $A[k]$, it doesn't matter if we swap them with $A[k]$, so they can remain where they are.
- This means that if $A[k]$ partitions A, we can just sort $A[1 \ldots k-1]$ and $A[k+1 \ldots N]$ independently, and then A will be sorted!
- This gets us to QuickSort! We implement QuickSort *in-place* — we don't use memory outside of the original array — and we use a recursive helper function to manage which sub-arrays we're currently sorting. You can see the recursive structure in Alg. 1, but we don't yet implement PARTITION, the algorithm that picks an element, puts it in $A[p]$, and then makes $A[p]$ partition A.

Algorithm 1 An in-place QuickSort implementation that uses a recursive helper function

```
function QuickSort(Array A)
    QuickSort(A, 1, N)
end function
function QuickSort(Array A, Integer low, Integer high)
    if low ≥ high then
        return
    end if
    p ←Partition(A, low, high)
    QuickSort(A, low, p − 1)
    QuickSort(A, p + 1, N)
end function
```

- This is where I note that QuickSort is not *a* sorting algorithm, but a *collection* of sorting algorithms that have the above template. There is one QuickSort algorithm for every partitioning algorithm.
- We'll focus here on the *Lomuto partitioning scheme* (Alg. 2), but the Activity gives you the idea behind another, more efficient partitioning scheme called the *Hoare partitioning scheme*.
- Observe that we have to choose the element that we want to use to partition the array. This element is typically called the *pivot*, and, in the partitioning algorithms we consider, is conventionally chosen to be the last element of the subarray. Conceptually, it may as well be random — we'll see why when we discuss runtime analysis.

Algorithm 2 The Lomuto partitioning scheme

```
function Partition(Array A, Integer low, Integer high)
    p ← high
    split ← low
    for i ← low . . . high − 1 do
        if A[i] ≤ A[p] then
            Swap(A[i], A[split])
            split ← split + 1
        end if
        Swap(A[p], A[split])
        return split
    end for
end function
```

1.1 Proof of Correctness

- To see quicksort is correct, we simply need to construct the right definition of correctness for Partition!
  - A partition algorithm is correct iff it returns p such that $A[p]$ partitions $A[low \ldots high]$.

- Let's assume PARTITION from above is correct and prove QuickSort correct first!

  Statement: After $QuickSort(A, low, high)$, $A[low \dots high]$ is sorted.

  Base Case: First let N be the size of the subarray we're sorting, and note that it is $N = high - low + 1$, so if $low \geq high$, $0 \leq N \leq 1$, which means $N = 0$ or 1. Of course, as we've seen time and time again, arrays of length 0 and 1 are, by definition, sorted. In this case, we immediately return, and thus we return with $A[low \dots high]$ sorted.

  Inductive Step: We will use strong induction, so we assume by our inductive hypothesis that $\forall 1 \leq N \leq k$, $QuickSort$ will be correct. We now need to show that for a subarray of size k, $QuickSort$ will sort the subarray.

  First, we call PARTITION on the full array, which, because we're assuming it's correct, will return p such that $A[p]$ partitions $A[low \dots high]$. This gives us that $\forall e \in A[low \dots p-1]$, $e \leq A[p]$ and $\forall e \in A[p+1 \dots high]$, $A[p] \leq e$.

  We then make two recursive calls to QUICKSORT. Observe that neither of these recursive calls has within the bounds of the subarray it recurses on. This means that the length of the subarray it recurses on is, at most, of size $k-1$, which means that by our inductive hypothesis, the recursive calls are correct! This means that $A[low \dots p-1]$ and $A[p+1 \dots high]$ are sorted, and thus

  $$A[low] \leq \cdots \leq A[p-1] \leq A[p] \leq A[p+1] \leq \cdots \leq A[high]$$

  The left chain of inequalities comes from the fact that $A[low \dots p-1]$ is sorted, the last chain of inequalities comes from the fact that $A[p+1 \dots high]$ is sorted. Then observe that $A[p-1]$ was, before the sort of the left half of the array, an element in $A[1 \dots p-1]$(the left hand size of the partition), and thus $A[p-1] \leq A[p]$ from the fact that partition is correct. A parallel argument gives us the last remaining inequality: Since $A[p+1]$ was, before the recursive sort, an element in $A[p+1 \dots N]$ (the right hand side of our partition), we have by the correctness of our partitioning that $A[p] \leq A[p+1]$. This completes the above inequality, and then observing that this is the same as saying $A[low \dots high]$ completes the proof.

- Now let's prove PARTITION correct! Since the algorithm is iterative, we must first construct a loop invariant. I recommend this as an exercise, so before reading on, step through the algorithm for a few examples and try and come up with the proof yourself!

  Loop Invariant: After the iteration of the for-loop where $i = k$, $\forall e \in A[low \dots split-1]$, $e \leq A[p]$ and $\forall e \in A[split \dots k]$, $e > A[p]$

  Base case: Before the loop, $i = low - 1$(one prior to the first iteration) and split is initialized to low, so $A[low \dots split-1]$ and $A[split \dots low-1]$ are both empty, and the loop invariant is trivially true: There are no elements to compare!

  Inductive step Assume that after the loop where $i = k$ $\forall e \in A[low \dots split-1]$, $e \leq A[p]$ and $\forall e \in A[split \dots k]$, $e > A[p]$. We must show that after the loop where $i = k+1$, $\forall e \in A[low \dots split-1]$, $e \leq A[p]$ and $\forall e \in A[split \dots k+1]$, $e > A[p]$

  Let's break this into two cases, based on whether we enter the if statement!

Case 1 If we don't enter the if-statement, we do nothing. Because nothing changed in the array, by assumption we know that $\forall e \in A[low \dots split-1]$, $e \leq A[p]$ and $\forall e \in A[split \dots k]$, $e > A[p]$. The only thing missing from our desired statement is to show that $A[k+1] >$

$A[p]$ (The last element in the post-split chunk of the array to handle in the second part of the invariant!). Since we didn't enter the if-statement and this is exactly the negation of the condition of the if-statement, we know this to be true, and we're done.

Case 2 If we do enter the if-statement, we know that $A[k+1] \leq A[p]$. We then swap $A[k+1]$ and $A[split]$. This means that after the swap, $A[split] \leq A[p]$. Since before the swap, $A[split] \in A[split \dots high]$, by our inductive hypothesis we know that after the swap $A[k+1] > A[p]$.

We then increment *split*. For clarity, let's call this new value of *split* $split_{post}$ and this prior value of *split* $split_{pre}$. $A[low \dots split_{pre} - 1]$ is untouched in the loop, so for $e \in A[low \dots split_{pre} - 1]$, $e \leq A[p]$. similarly, $A[split_{pre} + 1 \dots k]$ is untouched, so for $e \in A[split_{pre} + 1 \dots k]$, $e > A[p]$.

Converting these to $split_{post}$ terms, we have that $\forall e \in A[low \dots split_{post} - 2]$, $e \leq A[p]$ and $\forall e \in A[split_{post} \dots k]$, $e > A[p]$. If we compare this to what we need to show, all that's left unproven is that $A[split_{post} - 1] \leq A[p]$ and that $A[k+1] > A[p]$. Of course, those are the exact two things we concluded after the swap (see the first paragraph of this case!)

With the loop invariant true, we can conclude that after the final iteration where $i = high$, $\forall e \in A[low \dots split - 1]$, $e \leq A[p]$ and $\forall e \in A[split \dots high - 1]$, $e > A[p]$. This is so close to partitioning! We then swap $A[p]$ and $A[split]$. Then observe that p is defined just for clarity: By convention, $p = high$! So we're actually swapping $A[high]$ and $A[split]$. $A[split]$ before the swap was $> A[p]$, so after $A[high] > A[split]$. Because of the swap, now all relations w.r.t. $A[p]$ are now w.r.t. $A[split]$! This means that $\forall e \in A[low \dots split - 1]$, $e \leq A[split]$ (none of the elements in in this subarray changed!) and $\forall e \in A[split + 1 \dots high]$, $e > A[split]$ (here split became the *pivot* element and $A[high]$ became what used to be in $A[split]$). Now we just note that this is the definition of $A[split]$ partitioning $A[low \dots high]$ and thus returning *split* demonstrates that PARTITION is correct.

1.2 Runtime Analysis

1.2.1 Worst-Case

The worst-case time complexity is unfortunately fairly easy to see.

First, convince yourself that PARTITION is $\Theta(n)$

Suppose we're really unlucky with our pivots, and we always pick the largest element in the subarray. After each partitioning, we will split a subarray of size k into two subproblems of sizes... $k-1$ and 0. It will take $\Theta(n)$ recursive calls to reduce a problem of size n to our base cases of size 0 or 1, and thus our algorithm will be $\Theta(n^2)$. Not good! Worse than MergeSort and HeapSort, and just as bad as Insertion, Selection, and BubbleSort!

But wait....

1.2.2 Average-Case

This is a rare case where we have a substantially better average-case, but this means that we'll need to pull out some probabilistic reasoning.

Following Skiena, lets define a *good-enough* pivot as a pivot in the middle 50% of values. Formally, lets define the *rank* of an element in an Array as it's index in a sorted version of that array. A good enough pivot p has rank r such that $\frac{N/4}{\leq} r \leq \frac{3N}{4}$.

In an average-case analysis, we need to start talking about probabilities. What is an average case for sorting? Let's assume all orderings of the N elements in our array have equal probability. This means that there is a 50% chance that we get a good-enough pivot for every call to PARTITION.

What happens when we get a good-enough pivot? At worst, we divide a problem of size k into a problem of roughly size $\frac{3k}{4}$ and one of roughly size $\frac{k}{4}$. If we always follow the larger of the two subproblems down the recursion chain, we can find that deepest recursion chain. How many times do we recurse? Well if we multiply the size by roughly $\frac{3}{4}$ after each recursive call, we can solve for the max recursive depth $\lceil d \rceil$ by finding d that satisfies

$$(\frac{3}{4})^d n = 1$$
$$d = \log_{4/3} n$$

But wait, this is only true if we get a good-enough pivot at every call to PARTITION! This only happens 50% of the time! It's a coin flip!

Well now to get to the average part of average-case analysis. What is the *expected* (i.e., average) number of flips it takes to get d good-enough pivots?

Formally, we construct a random variables X_i such that $P(X_i = 1) = 0.5$ and $P(X_i = 0) = 0.5$, and define our count of good-enough pivots on a recursion chain k deep $D = \sum_{i=1}^{k} X_i$, and then ask what k gives us $E[D] = d$. Solving this out, we get

$$\begin{aligned}
E[D] &= d \\
E[\sum_{i=1}^{k} X_i] &= \log_{4/3} n \\
\sum_{i=1}^{k} E[X_i] &= \log_{4/3} n \\
kE[X_i] &= \log_{4/3} n \\
k(0.5(1) + 0.5(0)) &= \log_{4/3} n \\
0.5k &= \log_{4/3} n \\
k &= 2\log_{4/3} n
\end{aligned}$$

Of course, we could also just reason that if half of our coin flips come up heads, we should expect to get half of our flips to come up heads (a consequence of what probability folks call *linearity of expectation*), and thus after $2d$ recursions we should expect d good enough pivots.

Now we just observe that we do $\Theta(n)$ work at each recursive depth, and thus doing $\Theta(n)$ work $2\log_{4/3} n$ times is $\Theta(n \log n)$!

Note that here we assume that if we don't get a good-enough pivot, we do *no work at all*! This is a strict upper bound! What we've actually shown is that the average case of QUICKSORT is $O(n \log n)$. Of course, we've also assumed the worst when we said our good-enough pivot splits the array 75/25 instead of 50/50! This entire argument is only about upper bounds!

We could do a more sophisticated lower bound for the average case, but luckily, the Ω comes for free! It turns out, we can prove all comparison-based sorts (like all of them we've looked at so far!) are $\Omega(n \log n)$. Stay tuned on this... it'll come up later!

1.2.3 Average-Case *Inputs*

- Let's dig a little deeper into what we mean when we say *average-case analysis*.
- We adopt *-case analysis because for an input of size n, there are often multiple different inputs (i.e., for sorting, there are many different arrays of size n!). The best/worst-case analysis tells us for each n, we write the time complexity growth function for the best/worst *inputs* to our function (of size n).
- it's important to keep in mind that *-case analysis is about the kinds of inputs of size n, distinct from our conversations about big-O, Ω, Θ. One way to make this clear is to note that if we're doing this formally, picking a case to analyze lets us write growth functions, while choosing a big-O, Ω, Θ notation gives us a nicer way to write a growth function that we already have by letting us talk about that growth function in terms of asymptotic upper and lower bounds that are neater to write down (i.e., nice functions!)
- Average-case analysis is a little bit more complex that best/worst-case, because we consider the *average input*.
  - In practice, this analysis is probabilistic. We assume that we know a probability distribution over possible inputs and we define our growth function as the *expected* number of time steps as a function of n.
  - Since this course requires no background in probability (and it's tricky enough as it is!), we won't be diving deep into this kind of analysis, but know it's there!
- One consequence you should know is that when we talk about the average case analysis, we are talking about the average *input*. The execution of the algorithm isn't itself probabilistic (yet...), and so we can still have inputs that are particularly bad for QuickSort (i.e., inputs that are always in the worst-case).
  - For example, consider giving a sorted array as input to the QuickSort we showed above. What always happens with partition?
  - This is often inconvenient for someone just trying to pick the right sorting algorithm to use. Why should I have to think hard about the kinds of arrays I'll be feeding into my sorting algorithm. The arrays we sort aren't always uniformly distributed across every possible random ordering (an assumption we make in our average-case analysis! See if you can figure out where in comes in.)
- One clever trick Skiena gives us to solve this problem to fully commit to the randomization.
  - Before we run quicksort, spend $O(n)$ time to shuffle your input array.
  - Now, if you're shuffle algorithm is good, your input array's order will always be sampled uniformly at random from all possible orderings (i.e., the probability of our pivots rank being any value 1 to N is now actually equal no matter what the input is!)

– Now we've move from having average guarantees over all inputs to have expected (probabilistic!) guarantees for *all* inputs.
– i.e., no matter what the input order is, if we're not unlucky, we should expect $\Theta(n \log n)$ asymptotic time complexity.